Interaction Cost and Shotgun Profiling

BRIAN A. FIELDS and RASTISLAV BODIK
University of California, Berkeley
MARK D. HILL
University of Wisconsin, Madison
and
CHRIS J. NEWBURN
Intel Corporation

We observe that the challenges software optimizers and microarchitects face every day boil down to a single problem: bottleneck analysis. A bottleneck is any event or resource that contributes to execution time, such as a critical cache miss or window stall. Tasks such as tuning processors for energy efficiency and finding the right loads to prefetch all require measuring the performance costs of bottlenecks.

In the past, simple event counts were enough to find the important bottlenecks. Today, the parallelism of modern processors makes such analysis much more difficult, rendering traditional performance counters less useful. If two microarchitectural events (such as a fetch stall and a cache miss) occur in the same cycle, which event should we blame for the cycle? What cost should we assign to each event? In this paper, we introduce a new model for understanding event costs to facilitate processor design and optimization.

First, we observe that all instructions, hardware structures, and events in a machine can interact in only one of two ways (in parallel or serially). We quantify these interactions by defining *interaction cost*, which can be zero (independent, no interaction), positive (parallel), or negative (serial).

Second, we illustrate the value of using interaction costs in processor design and optimization. In a processor with a long pipeline, we show how to mitigate the negative performance effect of long latency "critical" loops, such as the level-one cache access and issue-wakeup, by optimizing seemingly unrelated resources that *interact* with them.

Finally, we propose *shotgun profiling*, a class of hardware profiling infrastructures that are parallelism-aware, in contrast to traditional event counters. Our recommended design requires only modest extensions to current hardware counters, while enabling the construction of full-featured dependence graphs of the microexecution. With these dependence graphs, many types of analyses can be performed, including identifying critical instructions, finding slack, as well as computing costs and interaction costs.

Categories and Subject Descriptors: C.4 [**Computer Systems**]: Performance of Systems—*Modeling*; *measurement*; *attributes*; D.2.8 [**Software**]: Software Engineering—*Metrics*

Authors' addresses: Brian A. Fields, Rastislav Bodik, Department of Computer Science, 517 Soda Hall, University of California, Berkeley, CA 94720-1176; email: bfields@berkeley.edu; Mark D. Hill, University of Wisconsin, Madison, WI; Chris J. Newburn, Intel Corporation.

General Terms: Performance, Measurement, Design

Additional Key Words and Phrases: Performance analysis, critical path, profiling, modeling

1. INTRODUCTION

Modern microprocessors achieve much of their performance through rigorous exploitation of fine-grain parallelism. The key dilemma caused by this parallelism is, *Which event are we to blame for a cycle that experienced two (or more) simultaneous events (for example, when a window stall and a multiplication occurred simultaneously)?* Clearly, both of these events must be optimized to remove the cycle, but how do we express this fact in a performance breakdown?

Another view of the overlap dilemma is to ask, *What performance monitoring hardware can I add to my processor to answer these questions?* Counting events, event latencies, or both also fails to capture overlap.

This paper argues that if we could answer the above questions without losing track of the microarchitectural parallelism, we would help the designer to resize just the right queue, predict the most critical dependence, or, conversely, economically reduce the sizes of nonbottleneck resources, saving area and energy. In short, we could build more balanced machines, where no resource is waiting on another.

We answer these questions with analysis that is simple, yet powerful enough to make sense out of simultaneous bottlenecks in complex machines. A *bottleneck* is any *set* of events that contribute to execution time, while the *cost* of a bottleneck is simply the speedup obtained from idealizing the bottleneck's events. How events are grouped into a *set* depends on the application of the analysis. For example, a software-prefetching optimization might consider the set of all cache misses from a single static load, while hardware designers might focus on all events pertaining to a resource (e.g., all branch mispredictions).

Cost is a powerful metric because it reveals how much an optimization helps before further improvement is stopped by a secondary bottleneck. Moreover, events with cost zero may be good targets for "deoptimization" (e.g., making a queue smaller without affecting performance).

This standard notion of cost, of course, tells us nothing about our simultaneous bottlenecks, as illustrated by the fact that the cost of each of two completely parallel cache misses is zero. As the first contribution of our paper, we define *interaction cost* (*icost*), which reveals how two (or more) events interact in a (parallel) microexecution. Specifically, interaction cost of two events a and b is the difference in speedup between idealizing both together ($cost(a, b)$) and the sum of idealizing them individually: $icost(a, b) \stackrel{\text{def}}{=} cost(a, b) - cost(a) - cost(b)$. That is, interaction cost quantifies the cycles that can be removed only by optimizing both events together. Analogously, we can define the interaction cost between sets of events (e.g., all cache misses interacting with all ALU operations) by replacing a and b with sets of events.

The second contribution of our paper is to explore the utility of interaction cost for everyday design practice. We find that, somewhat surprisingly, interaction costs can be zero (e.g., for two independent cache misses), positive (e.g., for

two parallel cache misses), and even negative (e.g., for two cache misses in series with each other but in parallel with other events).

A *zero interaction* cost between two (sets of) events implies that we can design and evaluate optimizations for the two in isolation, as the events are independent: optimizing one will not change the cost of the other.

A *parallel interaction* (i.e., positive *icost*) reveals that events overlap, which implies that there is speedup which can be gained only by optimizing both events (e.g., two cache misses that completely overlap).

A *serial interaction* (i.e., negative *icost*) means that two events are in series with each other, but also in parallel with some other event. It thus reveals that completely optimizing both events is not worthwhile; rather, one should target either only one or both partially. Serial interaction gives the designer flexibility to attack what is easiest to improve and eschew optimizing structures that are already too big, power-hungry, or complex.

Costs and interaction costs are most useful in practice if they can be efficiently measured in both simulation and hardware (e.g., with an extension to performance counters). They can obviously be computed by running many idealized and unidealized simulations. This approach, however, requires 2^n simulations for n events or resources, which may be too expensive if n is large.

For greater efficiency, we manipulate a microexecution dependence graph as an alternative to complete resimulation. This graph is similar to the one used in previous work [Fields et al. 2001, 2002; Tune et al. 2002]. It captures both architectural dependencies (e.g., data dependencies) and microarchitectural events (e.g., branch mispredictions).

Finally, to measure interaction costs on real hardware running "live" workloads, we show, as our third contribution, how hardware can sample an execution in sufficient detail to construct a statistically representative microarchitecture graph. We call this hardware a *shotgun profiler* because of its similarity to shotgun genome sequencing [Fleischmann et al. 1995]. The profiler has low complexity (of the order of ProfileMe [Dean et al. 1997]) and is suitable not only for measuring interaction costs, but also for accurately identifying critical instructions [Fields et al. 2001], measuring slack [Semeraro et al. 2002; Fields et al. 2002], or computing the simple individual costs [Tune et al. 2002]. Thus, it may serve as an alternative to the current hard-to-interpret performance counters.

2. ICOST: A UNIFYING NOTION OF PERFORMANCE ANALYSIS

As motivated in Section 1, determining the *costs* and *interaction costs* of events is essential to many forms of performance analysis. By defining interaction costs, this section deals with the effects of microarchitectural parallelism on the cost of events. To achieve uniform analysis, we use the term *event* to refer to any stall cause, whether due to data dependences, resource constraints, or microarchitectural events.

2.1 Cost

Intuitively, the cost of an event is not its execution latency, but its contribution to the overall execution time of the program. Equivalently, the cost is the execution

Table I. Idealizing Events

Event Type	How to Idealize in a Simulator
Icache, dcache misses	Turn misses into hits
ALU operation	Give ALU zero cycle latency
Fetch, Issue, Commit BW	Use infinite BW
Branch mispredict	Turn mispredicts into correct preds
Instruction window	Use infinite window

Listed are techniques to idealize a few of the events studied in this paper. Due to practical constraints (finite memory), we approximate an infinite window by using one that is 20 times larger than the baseline.

time decrease obtained if the event is *idealized*. Table I lists how some events can be idealized. Let e be an event, t be base execution time (nothing idealized), and $t(e)$ be execution time with e idealized. We formally define the cost of e, $cost(e)$ as

$$cost(e) \stackrel{\text{def}}{=} t - t(e)$$

The cost of an event can be naturally generalized to an *aggregate* cost of a set of dynamic events S. This allows us to compute, for example, the cost of a cache as the total speedup when all cache misses are idealized.

Observing the idealizations of Table I clarifies why this definition of cost is useful. A compiler seeking to prefetch load instructions would want to know how much execution time would improve if all dynamic cache misses from a single static load were idealized to hits. A hardware value predictor would want to know the improvement from idealizing particular data dependences. Finally, an architect considering enhancements to the instruction window would like to know how much such enhancements could improve performance.

2.2 Interaction Cost

While knowing the costs of individual events is useful, they are not always sufficient to drive optimization decisions. For instance, two completely parallel cache misses (c_1 and c_2) both have cost of zero ($cost(c_1) = cost(c_2) = 0$), since idealizing one would leave the overall critical-path length unchanged. Nevertheless, prefetching both loads may have substantial benefit.

Similar scenarios occur with analyses for making microarchitectural design decisions. For instance, an architect may find, via idealization, that the cost of cache load ports is low, suggesting it is not worthwhile to make the cache dual ported. The reality may be, however, that if the instruction window is also enlarged, increasing cache bandwidth could provide significant gain.

Essentially, the problem is that measuring the cost of individual events is only useful for determining "how critical" a *single* event is. In other words, standard cost gives no information about the content of "secondary" critical paths. While quantifying all secondary paths may seem a daunting task, we show below how to get a handle on the problem by measuring *interactions* between individual event costs.

Consider, for instance, the above example of the two cache misses. While the costs of the individual cache misses are zero, the *aggregate* cost of both cache

misses, obtained by measuring the execution time reduction from idealizing both c_1 and c_2 simultaneously, would be large. By knowing this aggregate cost, denoted $cost(\{c_1, c_2\})$, the program optimizer would know that while prefetching only one load would give little benefit, prefetching both would give significant benefit. We term this phenomenon, where $cost(\{c_1, c_2\}) > cost(c_1) + cost(c_2)$, a *parallel interaction*.

Perhaps less intuitively, it is also possible for the opposite parallelism-induced effect to occur, where $cost(\{c_1, c_2\}) < cost(c_1) + cost(c_2)$. One example is if two *dependent* cache misses, each with 100-cycle latency, both occurred in parallel with 100 cycles of ALU operations. In this situation, prefetching both provides no more benefit than prefetching either one alone, implying that a program optimizer would save overhead by performing only one prefetch. In general, this type of interaction can occur between two events A and B if they are in series with each other, but in parallel with some other event (or events) C. We call this phenomenon a *serial interaction*, since the two interacting events occur in series.

In summary, for two events e_1 and e_2:

$$\begin{aligned} cost(\{e_1, e_2\}) &= cost(e_1) + cost(e_2) \Leftrightarrow \text{Independent} \\ cost(\{e_1, e_2\}) &> \text{cost}(e_1) + cost(e_2) \Leftrightarrow \text{Parallel Interaction} \\ cost(\{e_1, e_2\}) &< cost(e_1) + cost(e_2) \Leftrightarrow \text{Serial Interaction} \end{aligned}$$

As we later empirically show, interactions are common phenomena (after all, there is potential for interaction any time two events occur simultaneously). To inform the optimizer (automatic or human) of the "degree" of interaction, we define interaction cost. Let e_1 and e_2 be two events and $cost(\{e_1, e_2\})$ be the aggregate cost of both events. Then, the *interaction cost* of e_1 and e_2, denoted $icost(\{e_1, e_2\})$, is defined as the difference between the aggregate cost of the two events and the sum of their individual costs:

$$icost(\{e_1, e_2\}) \stackrel{\text{def}}{=} cost(\{e_1, e_2\}) - cost(e_1) - cost(e_2)$$

Thus, for a parallel interaction, $icost(\{e_1, e_2\})$ is the number of extra cycles an optimization that targets both events, instead of just one, could ever hope to benefit. In contrast, for a serial interaction, $icost(\{e_1, e_2\})$ would be negative, reducing the expectation for performance improvement from targeting both events.

The interaction cost of two sets of events, S_1 and S_2, is defined similarly, by replacing e_1 and e_2 with S_1 and S_2 in the above equation. Moreover, the interaction cost of more than two events (or sets) can be defined recursively. Formally, let $\mathcal{P}(U) \setminus U$ denote the proper powerset of a set of events U (i.e., all subsets of U except for U itself). Then the interaction cost of U is defined as the cost of U minus the interaction cost of each proper subset of U:

$$\begin{aligned} icost(\{\}) &\stackrel{\text{def}}{=} 0 \\ icost(U) &\stackrel{\text{def}}{=} cost(U) - \sum_{V \in \mathcal{P}(U) \setminus U} icost(V) \end{aligned}$$

Finally, if U is the set of *all* events in an execution it follows that total execution time always equals the sum of the *icosts* for the powerset of U. This implies that completely accounting for execution time requires all interaction costs to be considered.

Interaction cost is a valuable tool for analyzing parallelism in out-of-order processors (and, potentially, parallel systems in general). Guiding load-prefetching decisions is only one example. The next section describes how to use interaction costs to construct parallelism-aware performance breakdowns, useful in making architectural design decisions.

2.3 Applying Icost: Parallelism-Aware Breakdowns

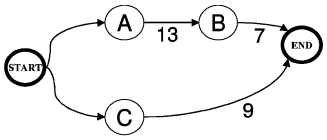
A *performance breakdown* of a microexecution answers the question, "how much do particular processor resources contribute to overall execution time?" Stated another way, a breakdown is a function that maps each cycle of execution to the events that are responsible for it. By allocating cycles among *base categories* of events (e.g., cache misses, ALU latencies, and the rest), a breakdown accounts for all cycles in the execution.

Traditional performance breakdowns (a.k.a., CPI breakdowns) map each cycle of execution delay to exactly one cause. This is fundamentally not possible in an out-of-order processor because sometimes multiple causes are to blame for a cycle. As a result, a traditional breakdown cannot accurately account for all cycles.

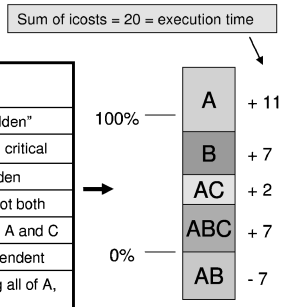
We improve traditional breakdowns by providing information about secondary critical paths. This approach enables an architect to determine when improving multiple resources will yield more benefit than an individual resource. Our solution is to have an explicit *interaction category* for each possible overlap among *base categories*. For example, if the base categories are data-cache misses (*dmiss*), ALU operations (*alu*), and branch mispredicts (*bmisp*), then there would be four interaction categories: $dmiss + alu$, $dmiss + bmisp$, $alu + bmisp$, $dmiss + alu + bmisp$. Each category would correspond to an interaction cost, similar to the example of Figure 1. With this representation, it is possible for a breakdown to account for all execution time.

Sometimes a graphical visualization, such as a stacked-bar chart, helps convey the messages contained in a breakdown. To account for interactions, however, the traditional stacked-bar chart must be modified—one possibility is illustrated in Figure 1(b). For the case study in this paper, we use tables to present results, since this representation is easier to comprehend when the number of categories is large.

Frequently Asked Questions (FAQ). Is it reasonable to define cost in a way that doesn't involve complete idealization (i.e., where the resource is made better but not perfect)? Yes, while the definition in terms of complete idealization given in Section 2.1 does meaningfully indicate how much execution time should be attributed to a particular resource, it may sometimes be more useful to define cost in terms of the maximum extent that a resource can be optimized. For instance, it may not be possible to get the effect of a zero latency ALU operation, but it may be possible to reduce the ALU latency by one cycle. In this case, the



Cat. Name	Expression for computing icost	Interaction Cost (icost)	Comments
A	cost(A)	11	A is partially "hidden"
B	cost(B)	7	All of B's latency is critical
C	cost(C)	0	C is totally hidden
AB	cost({A,B}) - icost(A) - icost(B)	11 -11 - 7 = -7	Optimize A or B, not both
AC	cost({A,C}) - icost(A) - icost(C)	13 - 11 - 0 = 2	Bonus for optimizing A and C
BC	cost({B,C}) - icost(B) - icost(C)	7 - 7 - 0 = 0	B and C are independent
ABC	cost({A,B,C}) - icost({A,B}) - icost({A,C}) - icost({B,C}) - icost(A) - icost(B) - icost(C)	20 - (-7) - 2 - 0 - 11 - 7 - 0 = 7	Bonus for optimizing all of A, B, and C



(a) Breakdown example **(b) One possible visualization**

Fig. 1. Correctly reporting breakdowns. (a) The traditional method for reporting breakdowns does not accurately account for *all* execution cycles, since it attempts to assign blame for each cycle to a *single* event when sometimes multiple events are simultaneously responsible. We propose a new method that uses *interaction costs*, discussed in Section 2.2. In this figure, the cost of an edge is the execution time reduction (or, in other words, reduction in critical-path length) obtained when the edge's latency is set to zero. Thus, $cost(A)$ is 11 because, after A is idealized (its latency set to zero), the critical path goes through C. The new critical-path length is 9, while the original length was 20, thus $cost(A) = 20 - 9 = 11$. (b) One possible compact visualization of this breakdown is shown. Here the positive interaction costs cause the stacked-bar chart to extend above 100%, but this is offset by negative interactions—which are plotted below the axis.

cost of the ALU could be defined as the execution time reduction obtained from decreasing the latency by one cycle. Fortunately, even with the new definition, our cost and icost equations can be applied unchanged.

3. MEASURING COST ON A DEPENDENCE GRAPH

Computing all costs and interaction costs for n sets (a.k.a., classes) of events can be done via 2^n simulations. Even if we restrict ourselves to only measuring interaction *pairs*, a quadratic number of simulations is required. Thus, a more efficient methodology than simulation is desired. Additionally, running multiple idealized simulations may not be possible for performance analysis on real hardware.

Our solution is to determine the effect of an idealization without actually idealizing the execution. We do this with a dependence-graph model of the microexecution where all the important events and resource constraints are modeled as latency-labeled edges. Then, for each idealization, we only need to alter a bottleneck's edges: by changing their latencies or by removing them. Since the graph does not contain *all* constraints in the machine, some precision is lost, but at a significant boost in efficiency.

We used the graph described in this section in two ways. First, in the case study of Section 4, it is used within a microprocessor simulator for improved efficiency in helping the hardware designer make better tradeoffs. Second, the graph is the enabling and necessary ingredient of our hardware profiling infrastructure, described in Section 5.

Table II. Constraints Captured by the Out-of-Order Processor Performance Model

Name	Constraint Modeled	edge	
DD	In-order dispatch	$D_{i-1} \rightarrow D_i$	
FBW	Finite fetch bandwidth	$D_{i-fbw} \rightarrow D_i$	where fbw is the maximum no. of insts. fetched in a cycle
CD	Finite re-order buffer	$C_{i-w} \rightarrow D_i$	w = size of the re-order buffer
PD	Control dependence	$P_{i-1} \rightarrow D_i$	inserted if $i-1$ is a mispredicted branch
DR	Execution follows dispatch	$D_i \rightarrow R_i$	
PR	Data dependences	$P_j \rightarrow R_i$	inserted if instruction j produces an operand of i
RE	Execute after ready	$R_i \rightarrow E_i$	
EP	Complete after execute	$E_i \rightarrow P_i$	
PP	Cache-line sharing	$P_j \rightarrow P_i$	inserted if inst. j produces cache miss to block loaded by i
PC	Commit follows completion	$P_i \rightarrow C_i$	
CC	In-order commit	$C_{i-1} \rightarrow C_i$	
CBW	Finite commit bandwidth	$C_{i-cbw} \rightarrow C_i$	where cbw is the maximum no. of insts. committed in a cycle

The meaning of the nodes are as follows: D, instruction dispatch into window; R, all data operands ready but waiting on functional unit; E, beginning execution; P, completed execution; C, committing. The constraints correspond to dependence edges in the graph. Operations are represented by latencies on the edges. An example instance of the dependence graph is shown in Figure 2.

The dependence-graph model. For our purposes, the graph model should meet two requirements: (1) idealizing on the graph should give the same speedup as in the simulator and (2) the analysis should be significantly more efficient than resimulation. Note that a dependence graph consisting of only *data* dependences would not meet the first requirement, for two reasons: (i) microarchitectural constraints (such as fetch bandwidth limitations) greatly affect cost calculations and (ii) we want to measure the costs of not just data dependences, but also hardware structures and events. Instead, we need a graph model that is microarchitecturally sensitive. Fortunately, previous work provides such a model [Fields et al. 2001; Semeraro et al. 2002; Tune et al. 2002], which we modestly enhance.

Our resulting graph provides a level of detail that reasonably meets both the accuracy and efficiency requirements given above (see Section 6 for an empirical assessment of its accuracy and the end of Section 4 for a discussion of efficiency). Table II describes the nodes and edges; and Figure 2 shows an instance of the model on a sample code snippet.

The graph model is essentially an annotated trace of the dynamic instruction stream. Each dynamic instruction i is represented by five nodes: the dispatch node D_i, the ready-to-execute node R_i, the start-execution node E_i, the completed-execution node P_i, and the commit node C_i. These five nodes represent events in the lifetime of the instruction's processing. Directed edges enforce ordering constraints (a.k.a., dependences) among these events. For instance, in Figure 2, the $P_0 \rightarrow R_1$ edge enforces the ordering that i_0 must complete execution before i_1 is ready to execute (a data dependence). Weights on the edges represent the latencies of operations. For instance, the weight of 10 on the $E_0 \rightarrow P_0$ edge is the execution latency of i_0; this edge provides the constraint that i_0 does not complete execution until 10 cycles after it starts execution.

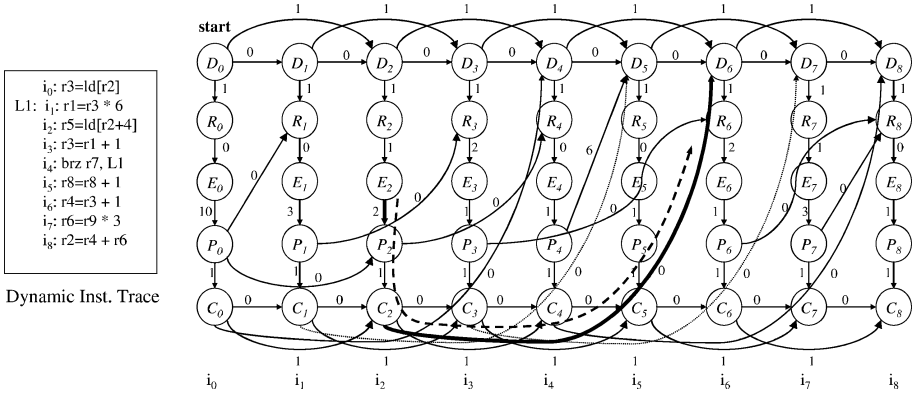


Fig. 2. An instance of the dependence-graph model from Table II. The dependence graph represents a sequence of dynamic instructions, assuming a machine with a four-instruction ROB and two-wide fetch/commit bandwidth. The dashed arrow shows how some load access *EP* edges and *CD* window edges are in series and, thus, have the potential to interact serially (see Section 4.1). Note that some other *EP* and *CD* edges are in parallel, thus there is also potential for parallel interaction between loads and the finite window constraint.

In this manner, all the important resource constraints in the machine are represented as dependence edges with weights representing latencies. For instance, the *DD* (and *CC*) edges (see Table II) constrain instructions so that they must be dispatched (and committed) in program order. The *FBW* (and *CBW*) edges limit the number of instructions that can be fetched (and committed) in a single cycle. In Figure 2, the fetch bandwidth is two instructions per cycle, so, for example, i_2 cannot be fetched until one cycle after i_0 is fetched; this constraint is represented by the one-cycle latency $D_0 \rightarrow D_2$ edge.

Similarly, the graph also models fetch stalls due to the reorder buffer (with *CD* edges) and branch mispredictions (with *PD* edges). Finally, cache-line sharing is modeled by placing an edge from the P node of any cache-missing load a to the P node of any subsequent load instruction b that accesses the same cache line. This *PP* edge prevents instruction b from *completing* execution until the cache miss is serviced by a. In this way, we model the effect of partial cache misses: if a is sped up due to an idealization, b may effectively change from a partial miss into a hit.

Measuring cost using the graph. As can be seen from the equations of Section 2.2, the interaction cost of two events $icost(a, b)$ is computed from several *simple cost* measurements: $cost(a, b)$, $cost(a)$, and $cost(b)$. In general, the *icost* of n events can be computed with $2^n - 1$ simple cost measurements. For measuring simple cost, we use the postmortem algorithm of Tune et al. [2002]. Essentially, their algorithm works by comparing the critical-path lengths of the baseline and idealized graphs—with several optimizations for efficiency.

FAQ. What are the modeling limitations of the dependence-graph method of computing cost? All of the complete idealizations listed in Table I can be easily implemented on the dependence graph by either removing the appropriate edges or setting their latency to zero. Furthermore, most notions of cost that

do not involve complete idealization can also be easily handled. For instance, if the cost of an ALU operation is defined as the speedup from decreasing its latency by one cycle, the "idealization" would involve reducing the ALU edge latencies as opposed to setting them to zero.

There are some limitations of what the graph is able to model, however. For instance, if the cost of the data cache is defined as the speedup from doubling its size, the graph could still be used, but a cache simulation would need to be run to determine which data-cache misses should be turned into cache hits. The graph is still useful in this case, since running a cache simulation is much easier and more efficient than a detailed timing simulation.

Does the dependence-graph model all resource constraints in the machine? No, modeling all resources would cause the graph to become very complicated, slowing its processing and making it more difficult to build. Instead, we model the resources that are most important for the task at hand, which, in this case, is measuring costs and interaction costs. As discussed in more detail in Section 6, the graph we use performs well at this task, with an average error of 8%.

Designing a graph model is a process of trial and error guided by intuition, as opposed to a systematic procedure, and there are often tradeoffs among complexity, efficiency, and understandability. For example, a close study of the five-node model will reveal that, for our machine configuration, the R node could be removed without affecting the model's accuracy (by adding the RE edge weight to the EP edge). We kept the R node because we believe it eases understanding.

Is the dependence graph required in order to compute interaction costs? No, the graph makes the computation more efficient and is needed for our hardware profiler (see Section 5), but it is not required for computing interaction costs in a simulator. For that purpose, running multiple idealized simulations is sufficient (see Section 6.1 for a description of this approach). The advantage of this multiple-simulation approach is that, despite being less efficient, it can be applied immediately, without going through the effort of implementing a graph analysis infrastructure.

4. ICOST TUTORIAL: OPTIMIZING A LONG PIPELINE

Several recent studies have found significant performance improvements possible by increasing the length of the processor pipeline. The improvement comes from increased clock frequency, but this improvement is unfortunately offset by the increasing latency of *performance-critical loops*. A loop is a feedback path in the pipeline, where the result of one stage is needed by an earlier stage. Three of the most critical loops include: (i) the latency of a level-one data-cache access, (ii) the latency to issue back-to-back operations (the issue-wakeup loop), and (iii) branch mispredictions [Sprangle and Carmean 2002; Hrishikesh et al. 2002; Hartstein and Puzak 2002; Borch et al. 2002].

In this section, we present a tutorial on using interaction costs, by showing how they can quickly provide insights into processors with long pipelines. Interaction costs show us how to mitigate the performance impact of critical loops.

Finally, we compare our icost analysis conclusions to those of a conventional sensitivity study.

4.1 The Level-One Data-Cache Access Loop

Let us assume that the circuit designers optimized the level-one data-cache access as much as possible, but nonetheless the latency was higher than expected, say four cycles instead of the typical one or two. The question now is: What is the *most effective* way to change the microarchitecture to mitigate the effect of the high latency? Would it help to: (a) enlarge the branch predictor; (b) increase the number of load ports; (c) increase the data-cache size; or (d) increase the fetch bandwidth? Certainly, these changes will reduce the cost of each of these resources (if they were on the critical path), but will they also reduce the cost of data-cache accesses?

What we are looking for is a choice of something other than data accesses to optimize that will indirectly reduce the cost of those accesses. Optimizing some resource such as fetch bandwidth certainly will not affect the latency of data accesses, but the optimization might cause some of the latency to be removed from the critical path (or, in other words, "hidden" or "tolerated" by the machine). In essence, we are looking for *serial* interactions, since any resource that serially interacts with data accesses provides us an alternative resource for optimization that will enable us to remove the same set of cycles.

In our case study, before computing the interaction costs, we hypothesized what the outcome of the analysis could be, which amounted to predictions of where *serial* interactions would occur. We thought data dependences between data-cache-missing loads or ALU operations and level-one data-cache accesses might cause such a serial interaction. Another possibility would be an interaction between branch mispredicts and data-cache accesses, since loads often feed branches.

The results of the analysis is shown in Table III(a) (simulator parameters are in Table VII in Section 6). For brevity, the breakdown presents only those interaction costs that involve data-cache accesses, labeled 'dl1' in the table. In total, there would be $2^8 - 1 = 255$ costs and interaction costs if all of them were shown.

Before examining the correctness of our hypotheses, let us attempt to gauge the importance of interactions in general. If we sum up the singleton costs, say for *crafty*, we get a very high value, $24.5 + 16.3 + 6.0 + 16.4 + 6.7 + 11.3 + 0.8 + 0.6 = 92.6\%$. Does this mean interactions are only important for a small portion of the execution time, for example, 7.4% for crafty? The answer is "no," since these singleton costs could be counting the same cycles multiple times—in other words, serial interactions (negative icosts) may exist. In fact, the sum of the singleton costs for *vortex* is over 100, at 104%, which is only explainable by serial interactions. As expected, *vortex* does have interactions (in fact, large ones), both parallel and serial (and this is seen even when only considering interactions including *dl*1). So, we cannot make conclusions on the importance of interactions by looking at singleton costs alone.

Table III. Breakdowns for Optimizing a Long Pipeline

Category	bzip	crafty	eon	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
dl1	**22.3**	**24.5**	**17.5**	**13.6**	**18.6**	**31.9**	**7.4**	**19.1**	**31.6**	**19.1**	**28.5**	**19.8**
win	14.6	16.3	16.5	34.7	12.6	4.5	4.6	15.9	5.8	22.3	39.4	21.2
bw	4.0	6.0	6.1	2.2	5.6	5.3	0.5	2.7	7.2	3.4	4.8	4.2
bmisp	37.8	26.4	14.3	11.8	24.6	24.0	25.3	15.5	36.0	22.6	1.6	23.3
dmiss	21.8	6.7	0.7	21.8	25.1	8.1	78.8	30.4	1.4	32.5	19.7	31.1
shalu	9.8	11.3	4.9	12.5	5.3	20.6	1.4	17.3	7.3	7.5	5.4	7.8
lgalu	0.3	0.8	11.5	5.8	0.3	0.5	0.0	0.1	0.8	4.0	1.5	3.8
imiss	0.0	0.6	9.0	1.2	2.1	0.1	0.0	0.1	5.1	0.0	3.3	0.0
dl1 + win	**−5.2**	**−11.9**	**−7.9**	**−6.9**	**−3.9**	**−10.5**	**−0.1**	**−6.6**	**−5.6**	**−4.2**	**−25.9**	**−6.7**
dl1 + bw	5.9	10.1	7.3	3.0	11.4	5.6	0.3	4.6	9.2	1.5	16.8	2.2
dl1 + bmisp	**−9.1**	**−4.6**	**−3.7**	**−2.8**	**−6.3**	**−3.4**	**−2.3**	**−2.4**	**−7.3**	**−5.7**	**−0.2**	**−4.5**
dl1 + dmiss	−0.8	−1.2	−0.4	−0.5	−1.4	−1.0	−0.7	−1.8	−0.2	−2.3	−1.8	−2.4
dl1 + shalu	**−4.3**	**−4.6**	**−0.8**	**−2.1**	**−1.8**	**−12.7**	**−0.1**	**−4.8**	**−1.8**	**−0.4**	**−4.7**	**−2.0**
dl1 + lgalu	−0.3	0.2	−1.1	−0.6	−0.3	−0.5	0.0	−0.1	−0.7	−0.0	−1.3	−0.7
dl1 + imiss	0.0	0.0	1.1	0.3	0.3	0.0	0.0	0.0	1.1	0.0	0.6	0.0
Other	3.2	19.4	25.0	6.0	7.8	17.5	−15.1	10.0	10.1	−0.3	12.3	2.9
Total	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**

(a) CPI contribution breakdown (%) with four-cycle level-one cache

Category	gap	gcc	gzip	mcf	parser
shalu	**31.6**	**12.8**	**37.8**	**3.2**	**33.3**
win	39.8	11.0	11.6	3.8	16.7
bw	1.6	7.1	4.4	0.4	2.3
bmisp	7.9	26.0	23.9	27.3	13.6
dmiss	15.9	25.5	8.3	81.3	26.7
dl1	4.7	10.7	16.8	4.5	9.1
imiss	0.8	2.2	0.1	0.0	0.0
lgalu	4.3	0.5	0.6	−0.0	0.1
shalu + win	**−23.5**	**−2.0**	**−8.3**	**0.1**	**−11.7**
shalu + bw	9.9	9.9	8.7	0.7	6.1
shalu + bmisp	**1.2**	**−5.7**	**−5.3**	**−2.3**	**−1.3**
shalu + dmiss	1.4	0.1	−1.5	0.3	−0.4
shalu + dl1	**−0.9**	**−2.4**	**−10.5**	**−0.2**	**−4.0**
shalu + imiss	0.3	0.2	0.0	0.0	0.0
shalu + lgalu	−2.3	−0.4	−0.4	0.0	−0.0
Other	7.3	4.5	13.8	−19.1	9.5
Total	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**

(b) Breakdown with two-cycle issue-wakeup loop

Category	gap	gcc	gzip	mcf	parser
bmisp	**11.5**	**25.2**	**27.7**	**26.5**	**16.7**
dl1	6.6	10.3	19.0	4.5	10.3
win	36.2	12.8	16.3	3.7	11.3
bw	3.8	12.7	8.0	0.5	4.0
dmiss	25.7	28.8	10.8	81.0	30.0
shalu	11.9	4.9	21.0	1.4	17.3
lgalu	5.0	0.3	0.5	0.0	0.1
imiss	1.3	2.6	0.1	0.0	0.1
bmisp + dl1	−1.8	−4.6	−2.4	−1.5	−1.8
bmisp + win	**7.6**	**18.5**	**12.2**	**62.5**	**33.7**
bmisp + bw	−1.1	−1.1	−2.5	−0.2	−1.3
bmisp + dmiss	**0.3**	**−1.3**	**−0.1**	**−16.2**	**−2.5**
bmisp + shalu	0.7	−3.0	−3.7	−1.1	−0.8
bmisp + lgalu	0.3	0.0	0.2	−0.0	0.0
bmisp + imiss	−0.2	−0.1	−0.0	−0.0	−0.0
Other	1.3	−4.2	−11.7	0.5	5.9
Total	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**

(c) Breakdown with 15-cycle branch mispredict loop

Interaction costs are presented here as a percent of execution time and were calculated using the dependence graph in a simulator. The categories are: 'dl1' → level-one data-cache latency; 'win' → instruction window stalls; 'bw' → processor bandwidth (fetch, issue, commit bandwidths); 'bmisp' → branch mis-predictions; 'dmiss' → data-cache misses; 'shalu' → one-cycle integer operations; 'lgalu' → multi-cycle integer and floating-point operations; and 'imiss' → instruction-cache misses. Due to space constraints, only a subset of the SPECint bechmarks are shown for (b) and (c), but the benchmarks shown are representative of the suite. Note that 'Other', denoting the sum of all interaction costs not displayed, can be negative since the interaction costs can be negative. The machine moldeled is described in Section 6.1.

In analyzing the data, notice first that data-cache accesses have a large singleton cost, typically contributing 15–25% of the execution time. This means that 15–25% of the execution time would be eliminated if the data-cache access latency was reduced to zero. As for the interactions, we see that some of our hypotheses were correct: for instance, there are significant serial interactions between data-cache accesses and ALU operations (*dl*1 + *shalu*), suggesting we could mitigate the long data-cache loop by reducing ALU latency (perhaps

through value prediction [Lipasti and Shen 1996; Calder et al. 1999] or instruction reuse [Sodani and Sohi 1997]).

However, other conclusions from the analysis were not predicted beforehand. For example, it was hypothesized that large serial interaction might exist between data-cache misses and data-cache accesses. In reality, this interaction is very small: reducing data-cache misses is unlikely to mitigate the effect of the high-latency data-cache loop.

We also see that the largest serial interaction for most benchmarks is with instruction window stalls. Thus, perhaps the most effective mitigation of the data-cache loop would be to increase the size of the instruction window—a result that may be difficult to predict before performing the analysis.

Also, note that the *magnitude* of the interactions vary significantly across benchmarks. This variability suggests that interaction costs could be useful in workload characterization: their magnitude gives a designer early insights into what optimizations would be most suitable for the most important workloads.

Another way to view interaction-cost analysis is as an indication of where *imbalances* exist in the machine. For instance, consider the cache-access/window serial interaction mentioned above. The effect on costs and interaction costs when the size of the window is increased from 64 to 256 is shown below (for *vortex*).

	vortex		
	64	128	256
dl1	28.5	9.8	4.3
win	39.4	21.3	13.6
dl1 + win	**−25.9**	**−8.14**	**−2.7**
Exe Time	*100.0*	*80.8*	*75.0*

Notice how increasing the window size *reduces* the cost of the individual resources but *increases* the interaction cost. (In this case, increasing the icost causes it to become less negative.) These results indicate that the critical path is less dominant when the window size is larger—that is, the parallel "secondary" paths are closer in length to the critical path. Thus, the machine has become more *balanced* (i.e., there is less slack in the execution). More work is needed to fully explore whether (and if yes, how) interaction costs can be used to quantify to what extent a machine is balanced.

4.2 The Issue-Wakeup and Branch Mispredict Loops

We also performed the same analysis for the issue-wakeup and branch misprediction loops. In this section, we will highlight the results of the analysis.

The issue-wakeup loop. Suppose that a long pipeline demanded a two-cycle issue-wakeup latency, instead of the typical one. This will, of course, reduce performance, since ALU operations will not be able to issue back to back. Can we use serial interactions to determine how to mitigate the performance loss?

From the breakdown of Table III(b), we see significant serial interactions between ALU operations and several event classes: window stalls, branch mispredicts, and level-one cache accesses. The most significant interaction is, again, with window stalls; it is as large as −24% for *gap*. Because of this serial interaction, increasing the window size is more beneficial when the issue-wakeup latency is higher. For instance, we found that the speedup for *gap* when the window size is increased from 64 to 128 is 12% if the issue-wakeup latency is one and 18% if the latency is two, a difference of 50%.

The negative interaction costs also reveal for which benchmarks it is *not* going to be possible to mitigate the effect of longer pipeline loops by optimizing other parts of the machine. This is the situation in *gcc*, which exhibits very little serial interaction.

The branch misprediction loop. Finally, we consider the branch misprediction loop. Can we modify the microarchitecture to reduce branch misprediction costs? How about increasing the window size? Will that work to reduce branch misprediction loop cost in the same way it did for the other two loops?

The interaction costs in Table III(c) reveal that the answer is no. Instead of a serial interaction, there is a *parallel* interaction between branch mispredictions and window stalls. This parallel interaction tells us there are a significant number of cycles that can be eliminated only by optimizing *both* classes of events simultaneously. In other words, reducing window stalls is not likely to significantly reduce branch misprediction costs.

For a couple of benchmarks, *mcf* and *parser*, we do see significant serial interactions with data-cache misses (*dmiss*), however. In particular, for *mcf*, the serial interaction of −16.2% tells us that as much as 60% of the cost of branch mispredictions ($16.2/26.5 \times 100\%$) could be eliminated through optimization of data-cache misses. Intuitively, this effect is likely due to cache-missing loads providing data that is used to determine a branch direction. Again, interaction costs help: we can quantify the importance of this effect for particular workloads, even determining the static instructions where it occurs, helping to guide prefetch optimizations.

4.3 Comparing with Sensitivity Study

A sensitivity study is an evaluation of one or more processor parameters made by varying the parameters over a range of values, usually through many simulations. Interaction costs can be viewed as a way to *interpret* the data obtained from a sensitivity study. Regardless of how they are computed, through multiple simulations or graph analysis, interaction costs explain *why* performance phenomena occur in a very *concise* way.

Let us explore this relationship by validating that the conclusions obtained from interaction-cost analysis and conventional sensitivity studies are the same. We perform the comparison by using a corollary of the serial interaction between the instruction window and load latency (the main result of Section 4.1). As the load latency becomes larger, increasing the size of the instruction window has increasing benefit. Since load latencies and window stalls occur in series with each other (because *EP* edges are in series with *CD* edges,

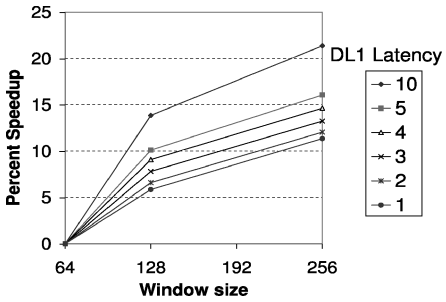


Fig. 3. Speedup from increasing window size for different level-one cache latencies. As predicted from the negative interaction cost, increasing the window size has a larger benefit when level-one cache latencies are larger.

as can be seen in Figure 2), increasing the latency of one will make both more dominant on the critical path.

Using this corollary, we performed the comparison by running several simulations to observe the speedup with increasing window size at different cache latencies (see Figure 3). Indeed, the interaction costs correctly predicted what the sensitivity study reveals: for instance, 50% greater speedup $((9-6)/6 \times 100\%)$ is obtained from increasing the window size from 64 to 128 when the data-cache latency is four instead of one.

From this example, we see the relationship between the two types of analyses. A full sensitivity study provides more information, for example, whether the curves in the plot are concave or convex; but interaction costs provide easier *interpretation* and concise *communication* of results. The interpretation is easy, since the type and magnitude of the icosts have well defined meanings. The ease in communication comes from the ability to summarize a large quantity of data very succinctly. For example, the entire chart of Figure 3 can be summarized by simply stating that the two resources interact serially. Furthermore, due to the formulaic nature of interaction cost, the interpretation is available *automatically*, without the effort of a human analyst.

Summary. In this section, we showed that interaction costs can help microarchitects during the design process. When the dependence graph is constructed by the simulator, architects can use interaction-cost-based breakdowns as a standard output of each simulation run. The overhead of building the graph during simulation in our research prototype is approximately twofold slowdown, which we did not find overly burdensome, considering the substantial benefit of the added insight. Furthermore, using the same principles of sampling that facilitate the profiling solution of Section 5, we found that the overhead could be reduced to approximately 10% without significantly impacting accuracy (with only 1–2% error due to sampling).

Perhaps even more exciting, however, is that all of this analysis can also be performed on real, deployed systems where resimulation and idealization is not an option. Hardware support for such analysis is the subject of the next section.

FAQ. If all icosts in a breakdown are zero, does that imply that there are no interactions? No, whether the interactions that exist show up in a breakdown depends on the choice of categories. Fundamentally, however, if any micro-operations occur in parallel, some interactions do exist. For instance, a pipelined processor that executes one instruction while another is being fetched does have, at least, a parallel interaction between these two micro-operations.

There are two ways that the choice of categories can cause interactions to be hidden. First, they may be too *coarse grained*, that is, contain too large of a set of events. For instance, there may be small interactions between ALU operations and memory stalls, but there may be large interactions between various types of memory stalls, for example, level-one and level-two cache misses. Second, the interactions may be *compensating*, in which serial and parallel interactions might "cancel" each other out. For instance, there may be both serial and parallel interactions between different pairs of individual ALU operations and memory stalls. The serial interactions will cause the icost between the two categories to decrease while the parallel interactions will cause it to increase. The final value, then, could be close to zero despite the existence of many interactions. This cancellation effect can be eliminated by either (a) measuring over smaller intervals of dynamic instructions, where only one type of interaction exists; or (b) using more fine-grained categories, so that only one type of interaction occurs between any pair of categories (e.g., if, hypothetically, ALU operations interact parallely with level-one cache misses and serially with level-two cache misses, splitting the "memory stalls" category could eliminate the compensatory effects.)

If there are no interactions, is the analysis useless? No, determining that interactions do not exist between resources can be as useful as finding out they do exist. The reason is that the analyst would then know that the resources are independent, and thus could each be optimized in isolation. This information can greatly simplify optimization tasks. For instance, if fetch bandwidth and ALU operations were found to be independent, a power-efficient processor could resize each resource in isolation, without worrying about any inefficiencies due to not making the optimization decisions jointly.

Does a larger cost value necessarily imply a greater execution time reduction when the resource is improved (e.g., enlarged, made faster, and so on)? No, if the improvement made to the resource is less than complete idealization, it is not necessarily true that a larger cost implies greater benefit from the improvement. A sensitivity study would be needed to provide this information (see Section 4.3). In our empirical experience, however, a large cost has been a good indication that improving the resource will produce a large reduction in execution time.

5. MEASURING COST IN HARDWARE: SHOTGUN PROFILING

In this section, we show how to measure icosts on real machines running real workloads. Our solution is to enhance existing performance counters so that they are sufficiently informative to construct a dependence graph from the data collected. This dependence graph is constructed offline in software, to keep

Table IV. Profiler Designs

Design	Problem	Solution
1. Hardware-intensive measurement	Hardware too expensive since it generates information too rapidly	*Sample* instructions sparsely
2. Sample each static instruction once	Does not distinguish between different microarchitectural behavior (e.g., an iteration of a loop with a branch misprediction versus an iteration without one).	Use microarchitectural context (in the form of a signature)
3. Record short microarchitectural signature around each sampled instruction	Accumulates error as each instruction is stitched together to form a graph	Use *long* signature that spans length of graph
4. Record long signature as a baseline and patch in sample profiles using short signatures		

Design #4, with both long and short signatures, is our final, recommended design.

hardware costs as low as possible. With the graph, icosts can be computed in precisely the same manner as they are in a simulator, as presented in previous sections. Furthermore, the graph can be used for other types of analyses as well, including identifying the critical path [Fields et al. 2001], finding slack [Fields et al. 2002], and computing single costs [Tune et al. 2002].

For purposes of understanding the design space involved, we explore a *series* of designs, summarized in Table IV. We start with an accurate but overly expensive solution and work towards a better tradeoff. Our goal is to keep hardware costs at a minimum while collecting sufficient information to construct the graph offline. The final and recommended solution is design #4 of Table IV.

5.1 Design #1: The Hardware-Intensive Approach

The conceptually simplest design would be to collect detailed latency and dependence information for every dynamic instruction as it flows through the machine, as is done in a simulator. The detailed information would be enough to construct all of the nodes and edges for each dynamic instruction, such that software could easily construct the graph offline. The exact information required will depend heavily on the processor implementation. For the simulated processor used in this paper, the information in Table V is sufficient.

Although this approach would be as accurate as constructing the graph in the simulator, it is not reasonably implementable. The primary reason is that the *density* of information collection is too great, in that too much data need to be collected simultaneously. To measure just one latency for every instruction would require a counter for each instruction in the machine at any one time—and to collect all of the information in Table V, many such latencies would need to be measured. Furthermore, moving all of this information through the machine would require many wires, affecting the bit pitch.

Table V. How Dependences and Latencies are Collected when Constructing the Graph

Dependence	col	Latencies	col
In-order dispatch (*DD*)	S	icache misses, itlb misses	D
Finite fetch bandwidth (*FBW*)	S	constant latency (1 cycle)	S
Finite re-order buffer (*CD*)	S	constant latency (0 cycle)	S
Control dependence (*PD*)	D	branch recovery latency	S
Execution follows dispatch (*DR*)	S	constant pipeline latency	S
Data dependences (*PR*)	reg: S, mem: D	constant latency (0 cycle)	S
Execute after ready (*RE*)	S	functional unit contention	D
Complete after execute (*EP*)	S	execution latency	D
Cache-line sharing (*PP*)	D	constant latency (0 cycle)	S
Commit follows completion (*PC*)	S	constant pipeline latency	S
In-order commit (*CC*)	S	store BW contention	D
Finite commit bandwidth (*CBW*)	S	constant latency (1 cycle)	S

'D' stands for dynamically, 'S' for statically. Dependences and latencies that must be determined dynamically are measured in hardware. Those that can be determined statically are inferred from the program binary (e.g., register data dependences) or the machine description (e.g., fetch and issue bandwidths). Besides the information above, a detailed sample also contains the PC of the instruction and the target address of indirect branches.

From this observation, we derive the most important constraint on the hardware: instructions should be profiled *sparsely*, in a sampling manner. For the rest of the designs, we assume hardware support for profiling only a *single* instruction, so that only one instruction can be sampled at a time. We chose this restriction because it is similar to current performance counter designs and proposals, for example, ProfileMe [Dean et al. 1997].

5.2 Design #2: One Sample per Static Instruction

A straightforward way to reduce the density of data collection is to maintain only one profile for each static instruction. With this restriction, randomly sampling one instruction at a time can quickly collect all the profiles that are needed. Then, instead of building the entire graph, *graph fragments* are constructed for hot program paths. The fragments are built by looking up the PC of each instruction in the hot path to find the corresponding static instruction's profile. The profile contains enough information to construct the nodes and edges corresponding to that instruction (since the profile contains all the information in Table V).

The primary advantage of this approach is that the requirements on the hardware data collection are substantially reduced, since only very sparse sampling is required. Unfortunately, however, the error is very high: empirically, the icosts computed are typically off by a factor of two or more when compared to those computed in the simulator.

The problem has to do with variations in microarchitectural behavior for different dynamic instances of the same segment of static code. As an example, consider Figure 4. In the first iteration of the loop, the instruction at PC 0×30 experiences an icache miss, while on the second iteration it does not. Thus, the graph for the first iteration is different than the graph for the second iteration, even though the same static code is executed (specifically, the DD edge latency is different).

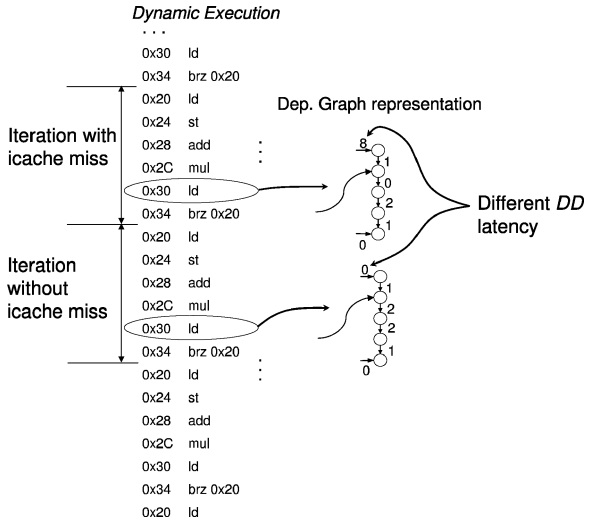


Fig. 4. Same static code, different microexecutions.

The obvious lesson here is that variations in the microexecution need to be distinguished in order to construct accurate graph fragments. In other words, multiple profiles for each static instruction need to be maintained. Specifically, we should ideally maintain one profile for each *microarchitectural context*, for example, in the example above, one sample for each instruction in both (a) an iteration with the icache miss and (b) an iteration without.

5.3 Design #3: Shotgun Profiler, Only Short Signatures

We distinguish between microarchitectural contexts by adding a *signature* to each sample collected from the hardware. The signature distinguishes between contexts by encoding microarchitectural events and state that surrounds the single dynamic "target" instruction. Thus, each sample consists of two things: (i) detailed latency and dependence information about the target instruction and (ii) a signature surrounding that instruction. If the signatures of two samples match, we assume the samples are from the same context.

The signature should uniquely identify the microexecution context while keeping the hardware cost as low as possible. More specifically, whenever the signatures for two samples are the same, the detailed latency and dependence information for the target instruction should also be the same. For our design, we chose to record two bits per dynamic instruction for 10 instructions before and after the targeted instruction. The two bits are an experimentally determined hash of microarchitectural context, specified in Table VI.

The graph-construction algorithm uses the signatures to determine which samples should be placed side-by-side within a graph fragment. As an example, consider Figure 5(a). Two samples are taken; in this case, they are of two different static instructions from two iterations of the same loop. By finding overlap among the appropriate signature bits between the two samples, we see that they "fit" together. Thus, they come from iterations of the loop with

Table VI. Description of Signature Bits

Bit	When to set to '1'
1	Set to 1 if the instruction is a (1) *taken* branch or (2) load or store. Reset to 0 if L2 dcache miss
2	Set to 1 if the instruction experiences a (1) L1 or L2 icache miss, (2) L1 or L2 dcache miss, (3) tlb miss, or (4) branch mispredict

The signature bits are meant to distinguish between different microarchitectural contexts. Experimentally, we determined the above hash function produced good results. Intuitively, the hash works well because it distinguishes between the most important events that occur in the microprocessor. For a different processor implementation than the one assumed in our simulator, a different signature might be required, perhaps one that uses more than two bits per dynamic instruction.

the same context and should be placed together in the graph fragment. By repeatedly applying this matching process, we can construct a graph fragment of arbitrary size.

This algorithm is very similar to a popular algorithm for DNA sequencing, called *shotgun sequencing* [Fleischmann et al. 1995] (see Figure 5(b)). Due to the similarity, we refer to the general class of profilers that use signatures as *shotgun profilers*. There is a large space of possible algorithms and infrastructures that exploit shotgun profiling, only a couple of which are presented in this paper.

Returning to the example of Figure 4, consider how a signature could help distinguish between loop iterations with different behavior. For the first iteration of the loop, an icache miss will appear in the signature; while in the second iteration it will not. Thus, the samples with the icache miss will be attached together in one portion of the graph fragment while the samples without the miss will be in another portion.

Empirically, we have found this design reduces the error by two to four times over one that does not distinguish between different microexecution behaviors. Nonetheless, the performance is still far from acceptable. The reason is that error accumulates for each sample placed into the graph, for a couple of reasons:

—*Missed correlation of distant events.* The context is only of nearby instructions, over a range of 20 instructions. If, for instance, the latency of an instruction is affected by an event that occurs 40 instructions away, this correlation cannot be captured. Since modern machines exploit parallelism across a rather large range of instructions, this effect can be significant.

—*Missing samples.* If an exact signature match cannot be found, the closest approximate match is used. In our experience, the missing samples are the ones with the rarest signatures, since they have the lowest probability to be collected. This causes rare events (e.g., branch mispredictions) to be underrepresented in the constructed graphs. Collecting more samples would reduce the error, but considering the exponential number of possible signatures, it may be infeasible to collect sufficiently many to eliminate the error.

To improve over this design, we need to reduce the accumulation of error. In the next section, we do this by adding a stable microarchitectural context "skeleton" on top of which the graph is constructed.

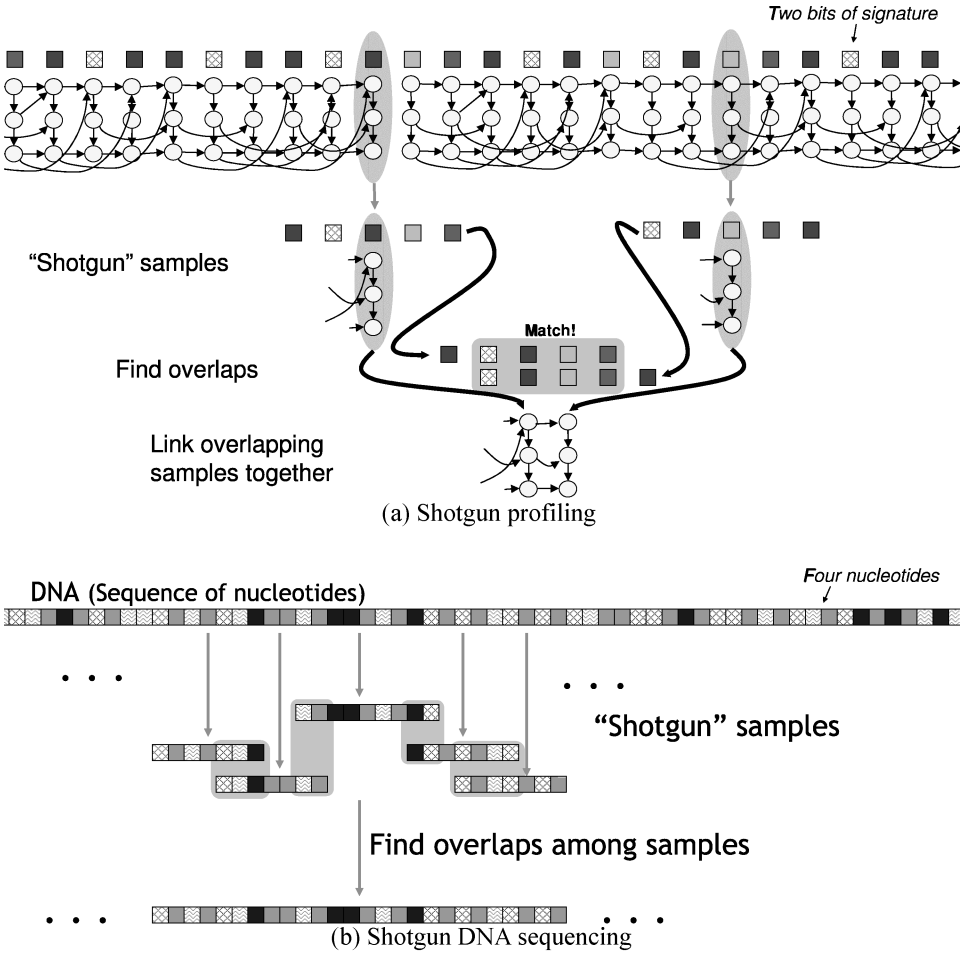


Fig. 5. Shotgun profiling and DNA sequencing. (a) The shotgun profiler works by collecting random "shotgun" samples that include a signature and detailed information about a single instruction. These samples are placed in a database and, offline, graph fragments are constructed by finding overlaps among the signatures of different samples. Our design uses a signature with two bits for each of the 10 dynamic instructions before and after the target instruction. For illustration, the figure uses a smaller signature. (b) DNA researchers face a problem similar to ours. Instead of constructing a graph, they seek to determine the sequence of nucleotides that comprise a strand of DNA. Their measurement apparatus, however, cannot simply observe the entire sequence at one time. Instead, they can only observe short, random, samples of the overall sequence. Their solution to this problem is called "shotgun" sequencing. First, many random samples are collected using their measurement apparatus. Then, offline, the full DNA sequence is constructed by looking for *overlaps* among the small fragments.

5.4 Design #4: Shotgun Profiler, Long and Short Signatures

Our final and recommended design introduces a second type of sample to be collected by the hardware, in addition to the one collected in design #3. The new sample is called a *signature sample* and consists of a single "start" PC and the two signature bits for each of the next 2000 dynamic instructions. Signature

samples are a natural way to identify correlation between distant events, and, as we shall show below, can also mitigate the effect of missing samples.

The software graph-construction algorithm works by first selecting a signature sample at random, which serves as a "skeleton" for the graph to be built. (The random selection ensures that each signature sample is chosen with equal probability, which naturally gives priority to hot microexecution paths.) The goal of the algorithm is to fill in this skeleton with *detailed samples* to form a latency-labeled dependence graph. A detailed sample is identical to the samples collected in design #3 above. To construct the graph, a detailed sample is selected for each dynamic instruction in the signature sample, where the selection is based both on a PC match and a signature match.

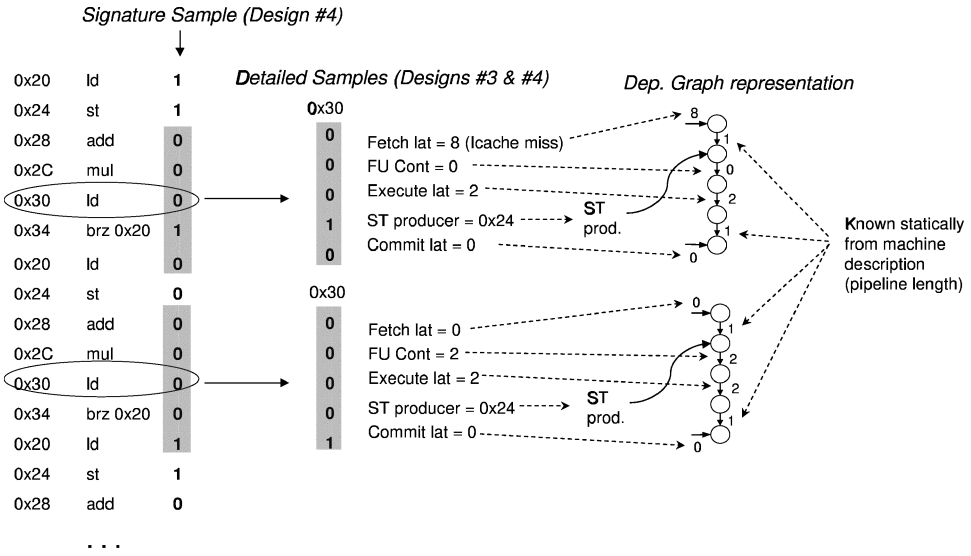
For example, consider building the graph nodes for the first instruction in the signature sample of Figure 6. The first instruction has PC of 0x24, so we look up detailed samples with this PC. Then, we select the one whose signature bits match the corresponding bits in the signature sample. Finally, the nodes for this instruction are constructed from the selected detailed sample.

If no detailed samples for the PC are found at all, which empirically happens less than 2% of the time, we infer what we can from the signature sample and the binary, using default values for unknown latencies. For example, if bit two of the signature is set to one and we know from the binary the instruction is a branch, we will infer that the branch was mispredicted. (In this instance, it is possible that an icache miss occurred instead of the branch mispredict, but we would guess a branch mispredict occurred for branch instructions.) Here, we see one advantage of the signature sample design over design #3: the signature sample gives us some information (e.g., whether a branch mispredict occurred) even when no matching detailed sample has been collected.

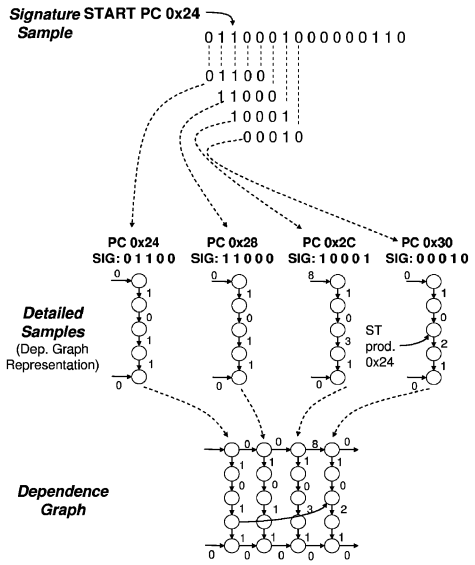
If some detailed samples are found, but none have an exact signature match, the detailed sample with the closest match is selected. An inexact match may reduce accuracy for that selection, but (unlike design #3) the signature sample provides a stable skeleton for future matches. Thus, a single mismatch does not cause error to propagate through the rest of the graph. The complete algorithm for constructing a graph fragment is in Figure 7.

5.4.1 *Determining PCs.* Remember that a signature sample consists solely of a start PC and the signature bits, i.e., to reduce hardware costs the PCs of other instructions are not recorded. Thus, we need to use some intelligence to infer the PC of each dynamic instruction in the signature sample. For direct conditional branches, we include the branch direction in the signature bits and lookup the binary for the target address of taken branches.

For indirect branches, we include the branch target address in the detailed samples. Assuming a signature match is a good indication of which target address an indirect branch will resolve to, the normal matching procedure described above will yield the correct next PC. We have found, empirically, that this procedure yields the correct target address most of the time, for 60–99% of the indirect branches, depending on the benchmark. (Note that this accuracy is highly dependent on the choice of signature; other signatures, perhaps using more bits, could achieve greater accuracy.)



(a) Hardware performance monitors



(b) Software graph construction

Fig. 6. The profiler infrastructure consists of two parts. (a) Hardware performance monitors. Our hardware performance monitors collect two types of samples: signature samples and detailed samples. For illustration, the figure shows one signature bit per instruction and collection of the bits for two instructions before and after each detailed sample. For greater accuracy, our design uses two signature bits per instruction (see Table VI) and collects signature bits for 10 instructions before and after each detailed sample (see Figure 7). (b) Postmortem software graph construction. The dependence graph is constructed by concatenating detailed samples, so that the resulting graph is representative of the microexecution denoted by the signature sample.

```
1.    Randomly select a signature sample for the skeleton.
      Call the starting PC in this sample the StartPC.
2.    For each instruction i from StartPC to end of fragment
2a.     Get from database all detailed samples with i's PC.
2b.     Select the detailed sample whose signature bits most closely
        matches the portion of the signature sample 10 instruction
        before i to 10 instructions after. The closeness of a match is
        judged by the number of identical bits.
2c.     Append sample's nodes and edges to the graph (see Fig. 6).
2d.     Determine PC of next instruction, i + 1 (call PC of i CurPC
        and PC of i + 1 NextPC):
2d1.      If i is not a branch, NextPC ← CurPC + 4
2d2.      If i is a direct branch and signature bit 1 of i is 1,
            Compute branch target and set NextPC equal to it
          Else NextPC ← CurPC + 4
2d3.      If i is a call, push target PC onto stack
          For returns, pop stack (if nonempty) and set NextPC to
          that PC
2d4.      If i is an indirect branch, set NextPC equal to target PC in
          detailed sample for i
2e.     Check for illegal signature bit/opcode combinations (see text).
```

Fig. 7. Algorithm for constructing a graph fragment in software.

In the cases where the matching sample's target address is not correct, there could be serious error in the graph fragment construction. To mitigate the error, we take advantage of the fact that some combinations of opcodes and signature bits could never occur down a correctly determined path. For instance, if an instruction on the long signature sample has its first bit set to one, it should be a load, store, or branch. If the computed PC (step 2d in the algorithm) does not correspond to one of these instruction types in the program binary, we know that there is an inconsistency and abort building the graph segment—building such a graph would lead to error in the results. We have found that 95–100% of errant graphs are indeed discarded using this technique.

Finally, note that for return instructions whose call counterpart occurs within the graph fragment, a stack of call addresses can provide the correct target address. If the call counterpart is outside the graph fragment, a return is treated the same as an indirect branch.

6. MEASURING PROFILER ACCURACY

In this section, we measure the accuracy of the shotgun profiler. We evaluate its accuracy by comparing the breakdowns it produces with the more accurate breakdowns produced from running multiple idealized simulations. We also break down the sources of error to understand better how the profiler could be improved in the future.

We find that the profiler's error in icost measurement is, on average, 9% off of the baseline, as measured via multiple simulations (methodology explained later). From the breakdown of error sources, we found that the modeling of the microprocessor as a dependence graph contributed more error than either the sparse sampling or the profiler algorithm.

Table VII. Configuration of Simulated Processor

Dynamically scheduled core	64-entry instruction window, 6-way issue, 15-cycle pipeline, perfect memory disambiguation, fetch stops at second taken branch in a cycle
Branch prediction	Combined bimodal (8k entry)/gshare (8k entry) predictor with an 8k meta predictor, 4k entry 2-way associative BTB, 64-entry return address stack
Memory system	32 KB 2-way associative L1 instruction and data (2-cycle latency) caches, shared 1 MB 4-way associative 12-cycle latency L2 cache, 100-cycle memory latency, 128-entry DTLB; 64-entry ITLB, 30-cycle TLB miss handling latency
Functional units (latency)	6 integer ALUs (1), 2 Integer MULT (3). 4 floating ALU (2), 2 Floating MULT/DIV (4/12), 3 LD/ST ports (2)

6.1 Methodology

We simulate the out-of-order processor described in Figure 7, using the SPEC2000int suite (as optimized Alpha binaries) with reference inputs. Our simulator is built upon the SimpleScalar tool set [Burger and Austin 1997]. We skipped eight billion dynamic instructions and then performed detailed timing simulation for 100 million.

We use the multiple-simulation approach as our baseline (Table VII). There is one simulation for each category in the breakdown where the simulation *idealizes* the appropriate set of event classes (see Table I in Section 2 for examples of idealizations). For example, for the category labeled "bmisp + dmiss," a simulation is run where (simultaneously) all branch mispredictions are made correct and all loads hit in the level-one cache.

6.2 Discussion of Category Errors

Table VIII shows breakdowns computed with the profiler relative to multiple simulations for the categories in Table III(a). A couple of observations can be made from the breakdowns. First, the type of interaction (parallel or serial) is always the same with the profiler as the *multisim* baseline. Second, the profiler comes very close to the *multisim* baseline most of the time, typically with error less than a few percent of the overall execution time.

There are some examples, however, where the error in the icost calculation is substantial. One category that tends to exhibit significant error for some benchmarks is the instruction window (*win*). For example, for *gap*, the error is −11.3% and for *vortex*, it is −8.4%. The cause of this error is the profiler's inability to completely accurately idealize the instruction window. Specifically, since the graph fragments constructed by the profiler are of finite size, it is not possible to accurately model a very large sized instruction window—needed when performing the idealization. Thus, the effective window size modeled by the profiler for idealization purposes will be smaller than that of the simulator, and thus it will likely under-predict the window's cost. This error could be reduced by increasing the size of the graph fragments constructed.

6.3 Sources of Error

In Table IX, we attempt to understand the sources of error in the profiler. To this end, the breakdowns of Table III(a) are computed in four different ways.

Table VIII. Measuring Accuracy of Profiler

	bzip			crafty			eon			gap		
	multisim	profiler	error	multisim	profiler	error	multisim	profiler	error	multisim	profiler	error
dl1	20.3	23.2	+2.9	23.4	24.2	+0.8	17.0	17.7	+0.7	12.6	12.6	+0.0
win	15.9	15.5	−0.4	17.3	15.4	−1.9	18.2	15.2	−3.0	**41.2**	**29.9**	**−11.3**
bw	**6.5**	**3.9**	**−2.5**	**8.7**	**6.7**	**−2.0**	**10.5**	**6.6**	**−3.9**	4.1	2.4	−1.7
bmisp	37.3	38.3	+1.1	26.0	24.1	−1.9	14.2	14.4	+0.2	11.3	11.4	+0.1
dmiss	23.3	23.5	+0.2	6.9	6.5	−0.4	0.8	0.6	−0.2	22.6	21.8	−0.8
shalu	8.9	10.0	+1.1	10.7	11.2	+0.5	4.5	5.2	+0.7	13.8	11.2	−2.6
lgalu	0.3	0.3	+0.0	0.7	0.8	+0.1	12.6	12.1	−0.5	5.3	5.7	+0.4
imiss	0.0	0.2	+0.2	0.7	0.2	−0.5	9.2	8.7	−0.5	1.3	0.9	−0.4
dl1 + win	−4.8	−5.2	−0.5	−11.5	−11.7	−0.2	−7.7	−7.2	+0.5	−6.3	−6.1	+0.2
dl1 + bw	6.9	5.9	−1.2	10.0	10.5	+0.5	6.9	6.8	−0.1	3.0	3.3	+0.3
dl1 + bmisp	−9.1	−9.6	−0.4	−4.9	−4.2	+0.7	−3.8	−3.9	−0.1	−2.9	−2.7	+0.2
dl1 + dmiss	−0.8	−0.7	+0.1	−0.4	−1.3	−0.9	−0.2	−0.3	−0.1	0.4	0.3	−0.1
dl1 + shalu	−3.5	−4.3	−0.8	−4.0	−4.5	−0.5	−0.6	−1.0	−0.4	−0.3	−2.1	−1.8
dl1 + lgalu	−0.2	−0.3	−0.1	0.3	0.2	−0.1	−0.5	−0.8	−0.3	−0.2	−0.5	−0.3
dl1 + imiss	0.0	0.0	+0.0	0.0	0.0	−0.0	1.3	1.0	−0.3	0.3	0.4	+0.1

	gcc			gzip			mcf			parser		
	multisim	profiler	error	multisim	profiler	error	multisim	profiler	error	multisim	profiler	error
dl1	17.4	17.0	−0.4	29.9	31.7	+1.8	**7.1**	**7.4**	**+0.3**	17.9	19.1	+1.2
win	14.4	13.0	−1.4	14.7	13.1	−1.6	4.8	4.3	−0.5	**17.1**	**13.2**	**−3.9**
bw	**9.0**	**7.1**	**−1.9**	6.6	5.5	−1.1	0.6	0.4	−0.2	4.0	3.0	−1.0
bmisp	23.9	21.5	−2.4	23.8	23.4	−0.4	25.3	25.1	−0.2	15.8	14.9	−0.9
dmiss	25.5	27.7	+2.2	8.1	7.8	−0.3	80.8	79.0	−1.8	32.1	28.1	−4.0
shalu	5.4	4.7	−0.7	18.9	20.7	+1.8	1.4	1.4	+0.0	17.9	17.1	−0.8
lgalu	0.6	0.2	−0.4	0.5	0.5	+0.0	0.0	0.0	+0.0	0.1	0.1	−0.0
imiss	2.1	1.4	−0.7	0.1	0.0	−0.1	−0.0	−0.0	+0.0	0.1	0.1	+0.0
dl1 + win	−4.1	−3.5	+0.6	−9.3	−9.6	−0.3	−0.0	−0.1	−0.1	−6.3	−6.2	+0.1
dl1 + bw	10.9	12.4	+1.5	6.2	5.7	−0.5	0.4	0.3	−0.1	4.9	4.9	−0.0
dl1 + bmisp	−6.3	−5.4	+0.9	−3.6	−3.1	+0.5	−2.3	−2.3	−0.0	−2.5	−2.4	+0.1
dl1 + dmiss	−0.9	−1.4	−0.5	−0.2	−1.3	−1.1	−0.4	−0.5	−0.1	−0.9	−1.7	−0.8
dl1 + shalu	−2.1	−1.4	+0.7	**−7.6**	**−9.4**	**−1.8**	−0.2	−0.1	+0.1	−4.1	−4.9	−0.8
dl1 + lgalu	−0.5	−0.2	+0.3	−0.5	−0.5	−0.0	0.0	0.0	−0.0	−0.1	−0.0	+0.1
dl1 + imiss	0.3	0.2	−0.1	−0.0	−0.0	+0.0	0.0	0.0	+0.0	−0.0	−0.0	+0.0

	perl			twolf			vortex			vpr		
	multisim	profiler	error	multisim	profiler	error	multisim	profiler	error	multisim	profiler	error
dl1	30.7	31.3	+0.6	**17.1**	**19.2**	**+2.1**	27.4	30.4	+3.0	18.5	20.3	+1.8
win	6.2	5.6	−0.6	24.2	22.3	−1.9	42.8	34.4	−8.4	22.9	21.9	−1.0
bw	10.3	8.1	−2.2	4.5	3.5	−1.0	**8.0**	**5.3**	**−2.7**	**5.9**	**4.4**	**−1.5**
bmisp	35.4	38.0	+2.6	22.2	22.6	+0.4	1.5	0.8	−0.7	23.4	23.1	−0.3
dmiss	1.3	0.8	−0.6	34.3	34.3	−0.0	19.8	18.7	−1.1	32.5	32.1	−0.4
shalu	7.4	8.2	+0.8	7.7	7.7	−0.0	3.9	5.4	+1.5	7.3	8.2	+0.9
lgalu	0.7	0.6	−0.1	4.2	4.2	+0.0	1.5	1.5	−0.0	4.1	4.0	−0.1
imiss	**5.3**	**2.7**	**−2.6**	0.1	0.0	−0.1	3.3	0.9	−2.4	0.0	0.0	−0.0
dl1 + win	−5.9	−5.4	+0.5	−3.6	−4.5	−0.9	−25.7	−27.0	−1.3	−6.2	−6.9	−0.7
dl1 + bw	9.9	9.7	−0.2	1.7	1.5	−0.2	17.7	17.7	+0.0	1.9	2.1	+0.2
dl1 + bmisp	−8.4	−8.2	+0.2	−5.8	−5.8	+0.0	−0.2	−0.1	+0.1	−4.6	−4.4	+0.2
dl1 + dmiss	−0.1	−0.1	−0.0	−0.1	−1.9	−1.8	−1.6	−1.2	+0.4	−1.4	−2.2	−0.8
dl1 + shalu	−2.2	−2.0	+0.2	−0.5	−0.3	+0.2	−3.3	−4.7	−1.4	−1.5	−1.9	−0.4
dl1 + lgalu	−0.7	−0.5	+0.2	−0.0	−0.1	−0.1	−1.2	−1.3	−0.1	−0.3	−0.6	−0.3
dl1 + imiss	1.0	0.6	−0.4	−0.0	−0.0	+0.0	0.5	0.1	−0.4	0.0	0.0	+0.0

Validation was performed on the same CPI contribution breakdown (with results expressed in percent of total CPI) as in Table III(a). The *multisim* column shows the value for each category computed through the multiple-simulation approach. This serves as the baseline for measuring accuracy. The *profiler* column shows the values the profiler computed, while the *error* column is the difference between the *profiler* and *multisim*. The single largest percent error (considering categories greater than 5%) for each benchmark is in bold.

multisim is the baseline, as above. *fullgraph* is the breakdown computed with the dependence graph of the entire program, just as was done for the results of Section 4. *graphfrag* is the breakdown computed assuming the graph fragments constructed by the profiler were perfect (i.e., exactly as they exist in the full graph), and *profiler* is the breakdown as computed on the imperfect

Table IX. Sources of Errors for the Shotgun Profiler

	bzip	crafty	eon	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
multisim→fullgraph	11.1	7.0	9.1	8.4	8.6	14.3	2.2	4.9	7.9	5.1	9.7	9.0
fullgraph→graphfrag	3.6	2.8	3.5	3.2	3.1	2.1	0.2	3.3	2.9	2.4	4.0	2.4
graphfrag→profiler	4.9	3.4	2.3	3.7	10.6	3.9	0.1	2.1	5.4	3.4	4.6	5.0
multisim→fullgraph	11.1	7.0	9.1	8.4	8.6	14.3	2.2	4.9	7.9	5.1	9.7	9.0
multisim→graphfrag	12.9	7.8	11.0	8.9	9.5	13.9	2.4	6.9	9.8	6.0	13.0	9.4
multisim→profiler	**11.1**	**7.8**	**9.5**	**8.9**	**11.7**	**9.3**	**2.5**	**9.0**	**12.6**	**3.7**	**12.4**	**9.2**

The breakdowns of Table III(a) were computed four ways to better understand the sources of error in the profiler. *multisim* is the breakdown computed via multiple simulations; it serves as the baseline for comparison. *fullgraph* indicates the dependence graph of the entire program was used, as in Section 4; *graphfrag* is the breakdown computed assuming the graph fragments constructed by the profiler were perfect; and *profiler* is the breakdown as computed on the imperfect graph fragments actually constructed by the profiler (described in Section 5). The numbers presented are the average percent difference in the categories (excluding categories under 5%) between the two schemes in the first column of each row. For instance, the *multisim→fullgraph* row is determined by computing *abs*(*multisim*−*fullgraph*)/(*multisim*) for each category over 5% and averaging the results. Note that the *multisim→profiler* row is the total error for the profiler.

graph fragments actually constructed by the profiler (using the signature-based algorithm).

The first series of measurements examines the accuracy of each step of the full profiling scheme. *multisim→fullgraph* is the error introduced by modeling the machine as a dependence graph, as opposed to using a detailed simulator. Typically, this error is less than 10%; but, nonetheless, it does often contribute the largest fraction of the overall error of the profiler. It can potentially be reduced by increasing the detail of the model to include currently unmodeled aspects of the microarchitecture, such as contention for memory busses.

The *fullgraph→graphfrag* row shows the error caused by measuring the breakdowns using only a relatively small number of graph fragments as opposed to the entire graph. This *sampling* error is a significant component of the overall error for some benchmarks, for example, *vortex*. The good news here is that this error can be reduced by simply running the program longer to collect more samples.

The *graphfrag→profiler* row shows the error introduced by the profiler's signature-based algorithm for constructing graph fragments. The error is due to two factors: (1) the signature not being sufficient to identify the correct detailed sample to paste into the graph and (2) a signature-matching detailed sample not being in the database. The second error factor can be reduced by simply collecting more samples, while the first requires some redesign of the signature bits.

For most benchmarks, the signature-based algorithm contributes only a modest amount to the error, typically less than 5%. An exception is *gcc*, with an error of 10.6%. Upon closer inspection, we found that this large error is primarily due to the target address of indirect branches not being determined correctly, leading to many graphs being discarded (see Section 5.4.1). One way to reduce the error would be to construct smaller graph fragments, so that the probability of encountering a difficult indirect branch in any one fragment is reduced. We found that reducing the fragment size from 2000 to 1000 reduced the error to 5.1% (but, averaged over all benchmarks, the larger size improved accuracy).

Another method would be to enhance the signature to improve its ability to distinguish indirect branch targets, for example, by adding an additional bit that is set equal to one of the bits of the PC.

The second series of measurements shows the error of three of the breakdown computations—*fullgraph*, *graphfrag*, and *profiler*—relative to *multisim*. The purpose of these measurements is to show how each individual source of error contributes to the overall error of the profiler. Notice that the overall error is not always monotonically increasing as each new source of error is included. For example, the *multisim→graphfrag* error for *eon* is 11.0%, while the *multisim→profiler* error is less, 9.5%. The reason is that the error introduced at each stage could be positive or negative, independent of the direction of errors at previous stages. Thus, it is statistically likely that the errors will compensate sometimes. In the case of the example, for *eon*, the *graphfrag→ profiler* error was mostly in the opposite direction of the errors in the previous two stages.

The overall error for the profiler is shown in the last row of Table IX, labeled *multisim→ profiler*. The range of errors for the benchmarks is from 3% (for *mcf*) to 13% (for *perl*), with the average error being 9%. Since the ability to compute costs and icosts from hardware profiles is qualitatively new, standards for accuracy have not been set; but an error of 9% seems small enough to perform meaningful analysis. If a smaller error is desired, increasing the precision of the graph model appears to offer the greatest opportunity for improvement.

7. APPLICATIONS OF THE HARDWARE PROFILER

Section 4 illustrated through a case study how interaction costs can be used to help microarchitects during the design process. For that use of our methodology, a simulator implementation is sufficient. Although a complete study of applications for the hardware profiler is beyond the scope of this paper, we outline in this section a few applications that seem to be naturally suited for the analysis the profiler enables. Recall that the dependence graphs constructed by the profiling infrastructure can be used to compute a variety of metrics, including criticality, slack, simple cost, as well as interaction cost.

Power-saving reconfiguration. One popular approach to reducing power and energy consumption is to resize hardware resources so that they just meet the needs of a particular application [e.g., Bahar and Manne 2001; Sasanka et al. 2002]. This resizing is beneficial, since smaller resources consume less power. There are several ways the hardware profiler can help:

—*Downsizing resources*. Once the graph is constructed from data collected by the profiler, a *slack* analysis can determine how much each resource can be decreased in size.

—*Upsizing resources.* Increasing the size of a resource (after it has been previously decreased) when the demands of the application become greater, has been identified as a difficult problem [Bahar and Manne 2001]. The profiler makes it easy, however, since if the *cost* of the resource is high, it would be logical to increase its size.

—*Interacting resources.* If two resources serially interact to a significant degree, it may be best to increase the size of only one of the two resources, since that will provide the bulk of the benefit. On the other hand, if there is a parallel interaction, both resources need to be increased. Finally, if the resources are independent (no interaction), they can be resized in isolation without loss of efficiency. Thus, determining that two resources are independent is very valuable, since policy decisions can be made in a *distributed* manner, without centralized control. (This distributed property is particularly useful for processors with multiple domains of dynamic voltage scaling on a single processor [Semeraro et al. 2002].)

Efficient preexecution. Preexecution is a proposal for a prefetching mechanism that constructs a *slice* of a program leading up to a performance-degrading event (e.g., a cache miss) and executes the slice earlier than it would through normal processing [Roth and Sohi 2000; Zilles and Sohi 2001; Collins et al. 2001]. With interaction costs, preexecution mechanisms can be made more efficient in two ways. First, if two loads have a large serial interactaction, preexecuting one load has most of the benefit of pre-executing both. Since the overhead of processing a pre-execution slice is relatively high, exploiting this characteristic could provide substantial benefit. For maximum overhead reduction, only the load associated with the shortest slice should be pre-executed.

Second, if two loads have a parallel interaction, both loads need to be pre-executed to get any benefit. If only simple costs were measured (without considering interactions), it is possible that neither of the two loads would be pre-executed, resulting in a lost opportunity.

Resource-aware code scheduling. Since the profiler produces a graph that contains information not only about the execution of instructions but also the machine's hardware resources, a compiler can perform more intelligent code scheduling than was previously possible. In particular, the compiler could exploit interactions to eliminate hardware resource bottlenecks. For instance, say a data-cache access serially interacts with a window stall (something that was found to occur empirically in Section 4). Since a serial interaction provides multiple options for eliminating the same set of cycles, scheduling the data-cache access earlier may reduce the cost of the window stall.

8. RELATED WORK

Previous work into microarchitectural performance analysis takes on many forms. Event counters and utilization metrics [Anderson et al. 1997; Zagha et al. 1996] have become standard and, before out-of-order processors, was all that was needed. When instructions execute in parallel, however, simply counting events is not enough to know their effect on execution time (e.g., two completely parallel cache misses cost the same as one, but a counter will report two). In response to the problems with counters, ProfileMe [Dean et al. 1997] supports pairwise sampling, where the latencies and events of two simultaneously in-flight instructions are recorded. With pairwise samples, one can determine the degree that two instructions' latencies overlap in time. Also,

the Pentium 4 [Intel 2003; Sprunt 2002] has a limited ability to account for overlapping cache misses. These performance-monitoring facilities do not appear amenable to computing a complete breakdown of execution time, however. We introduce interaction cost to provide this level of interpretability.

There are several works that aim to interpret the parallelism of out-of-order processors through fetch [Fahs et al. 2001; Patel et al. 1998] and commit attribution [Pai et al. 1997; Ranganathan et al. 1998; Rosenblum et al. 1995; Steffan et al. 2000; Rajwar and Goodman 2001; Hennessy and Patterson 2002], and at least one that combines attribution with some dependence information [Sasanka et al. 2002]. In these approaches, specific instructions and events are assigned blame for wasted fetch bandwidth or commit bandwidth, respectively. We have found that these analyses do, indeed, accurately compute the cost of certain classes of events, which was their intended purpose. They have not been used to compute interaction costs, however.

Several researchers have explored criticality and slack, two useful metrics for exploiting the parallelism in out-of-order processors [Srinivasan and Lebeck 1998; Fisk and Bahar 1999; Casmira and Grunwald 2000; Tune et al. 2001, 2002; Fields et al. 2001, 2002; Seng et al. 2001; Srinivasan et al. 2001; Semeraro et al. 2002; Rakvic et al. 2002]. Our notion of interaction cost extends these works by answering questions about nearly-critical paths, such as (i) "Which critical dependences are most important to optimize?" and (ii) "Which nearly-critical dependences should I optimize along with the critical ones?"

One of the above papers, by Tune et al. [2002], was the first to use the dependence graph to compute the cost of *individual* instructions in a simulator (we employ their algorithm). The focus of our paper is on how the costs of not only instructions but also machine resources *interact* in an out-of-order processor. We also provide a design for a hardware profiler, so that the analysis can be performed on real systems.

Karkhanis and Smith propose an analytical model for out-of-order superscalar processors [Karkhanis and Smith 2004]. The primary advantages of their model are its simplicity and ability to provide quick insights by evaluating analytical equations as opposed to resimulating (or performing a graph analysis). Its disadvantages include a lack of accounting for interactions and its specificity to out-of-order superscalar processors. In contrast, the interaction-cost analysis introduced in our work is applicable to any parallel system. The same is true of our shotgun profiler after suitable alterations are made to the graph model.

Note that Karkhanis and Smith confirm empirically that in the microarchitecture they study the interactions (called "overlaps" in their paper) of branch mispredicts and icache misses with dcache misses are relatively insignificant (in other words, that the resources are nearly independent). This discovery of near independence permits them to ignore interactions with a low, bounded error. For other resource classes or microarchitectures, interactions may be much more significant, as illustrated by the case study in this paper.

The MACS model of Boyd and Davidson [1993] assigns blame for performance problems to one of four factors: the machine, application, compiler-generated code, or compiler scheduling. They accomplish this by idealizing one factor at a time (to determine its cost). In comparison to this work, we focus

only on fine-grain microarchitectural events (as opposed to compiler decisions) and introduce a methodology for measuring interactions.

Yi et al. [2003] use a Plackett and Burman design to reduce the number of simulations required in a sensitivity study. However, their work does not quantify and interpret specific interactions between events. Allocation and analysis of variance (ANOVA) techniques do, in fact, quantify these interactions [Jain 1991]. ANOVA is inadequate for our purposes, however, for two reasons: (1) squaring of effects reduces their interpretability and (2) no distinction is made between positive and negative (parallel and serial) interactions.

9. CONCLUSION

The primary contribution of our work is establishing *interaction cost* as a methodology for bottleneck analysis in complex, modern microarchitectures. Interaction cost permits one to account for all cycles of execution time, even in an out-of-order processor, where instructions are processed in parallel.

We have also provided a relatively inexpensive hardware profiler design (close to the complexity of ProfileMe [Dean et al. 1997]) that enables measuring interaction cost in real systems. With this technology, not only microarchitects, but also software engineers, compilers, and dynamic optimizers can make use of the deeper understanding of bottlenecks.

For instance, feedback-directed compilers could favor prefetching cache misses that serially interact with branch mispredicts. Performance-conscious software engineers could identify the most important procedures and instructions for optimization and determine why the performance problems exist. Dynamic optimizers could save power by intelligently reconfiguring hardware structures. Finally, real workloads could be analyzed on real hardware, such as large web servers running a database.

ACKNOWLEDGMENTS

We thank Mary Vernon, David Wood, and Amir Roth for contributions to this work. We also thank Sarita Adve, Bradford Beckmann, Mark Buxton, Jarrod Lewis, David Mandelin, Milo Martin, Anat Shemer, Dan Sorin, Manu Sridharan, Renju Thomas, Min Xu, and the anonymous reviewers for comments on drafts of this paper. Finally, we thank the Wisconsin Architecture affiliates for feedback on early presentations of this work. This work was supported in part by National Science Foundation grants (CCR-0326577, CCR-0324878, CCR/CNS-0225610, EIA/CNS-0205286, CCR-0105721, EIA-0103670, EIA-9971256, CCF-0085949, CNS-0326577, CCF-0243657, and CDA-9623632), an NSF CAREER award (CCR-0093275), the UC MICRO Program, DARPA HPCS grant to IBM Corp., IBM Faculty Partnership Award, a Wisconsin Romnes Fellowship, and donations from IBM, Intel, Microsoft, and Sun Microsystems. Hill's sabbatical is partially supported by the Spanish Secretaría de Estado de Educución y Universidades. Hill has a significant financial interest in Sun Microsystems. Fields was partially supported by NSF Graduate Research and Intel Foundation Fellowships.

REFERENCES

ANDERSON, J. M., BERC, L. M., DEAN, J., GHEMAWAT, S., HENZINGER, M. R., LEUNG, S. A., SITES, R. L., VANDEVOORDE, M. T., WALDSPURGER, C. A., AND WEIHL, W. E. 1997. Continuous profiling: Where have all the cycles gone? *ACM Trans. Comput. Syst.*.

BAHAR, R. I. AND MANNE, S. 2001. Power and energy reduction via pipeline balancing. In *28th International Symposium on Computer Architecture*.

BORCH, E., TUNE, E., MANNE, B., AND EMER, J. 2002. Loose loops sink chips. In *8th International Symposium on High-Performance Computer Architecture*.

BOYD, E. L. AND DAVIDSON, E. S. 1993. Hierarchical performance modeling with MACS: A case study of the Convex C-240. In *20th International Symposium on Computer Architecture*.

BURGER, D. C. AND AUSTIN, T. M. 1997. *The Simplescalar Tool Set, version 2.0*. Tech. Rep., CS-TR-1997–1342, University of Wisconsin, Madison.

CALDER, B., REINMAN, G., AND TULLSEN, D. 1999. Selective value prediction. In *26th International Symposium on Computer Architecture*.

CASMIRA, J. AND GRUNWALD, D. 2000. Dynamic instruction scheduling slack. In *Kool Chips Workshop in Conjunction with MICRO 33*.

COLLINS, J. D., WANG, H., TULLSEN, D. M., HUGHES, C., LEE, Y., LAVERY, D., AND SHEN, J. P. 2001. Speculative precomputation: Long-range prefetching of delinquent loads. In *28th International Symposium on Computer Architecture*.

DEAN, J., HICKS, J. E., WALDSPURGER, C. A., WEIHL, W. E., AND CHRYSOS, G. 1997. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *30th International Symposium on Microarchitecture*.

FAHS, B., BOSE, S., CRUM, M., SLECHTA, B., SPADINI, F., TUNG, T., PATEL, S. J., AND LUMETTA, S. S. 2001. Performance characterization of a hardware mechanism for dynamic optimization. In *34th International Symposium on Microarchitecture*.

FIELDS, B., BODÍK, R., AND HILL, M. D. 2002. Slack: Maximizing performance under technological constraints. In *29th International Symposium on Computer Architecture*.

FIELDS, B., RUBIN, S., AND BODÍK, R. 2001. Focusing processor policies via critical-path prediction. In *28th International Symposium on Computer Architecture*.

FISK, B. R. AND BAHAR, R. I. 1999. The non-critical buffer: Using load latency tolerance to improve data cache efficiency.

FLEISCHMANN, R. D. ET AL. 1995. Whole-genome random sequencing and assembly of haemophilus-influenzae. *Science 269*, 496–512.

HARTSTEIN, A. AND PUZAK, T. R. 2002. The optimum pipeline depth for a microprocessor. In *29th International Symposium on Computer Architecture*.

HENNESSY, J. L. AND PATTERSON, D. A. 2002. *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann Publishers, Los Altos, CA.

HRISHIKESH, M. S., JOUPPI, N. P., FARKAS, K. I., BURGER, D., KECKLER, S. W., AND SHIVAKUMAR, P. 2002. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *29th International Symposium on Computer Architecture*.

INTEL. 2003. *Intel Pentium 4 Processor Manual*. Available at *http://developer.intel.com/design/pentium4/manuals/*.

JAIN, R. 1991. *The Art of Computer Systems Performance Analysis*. Wiley Professional Computing.

KARKHANIS, T. AND SMITH, J. E. 2004. A first-order superscalar processor model. In *31st International Symposium on Computer Architecture*.

LIPASTI, M. H. AND SHEN, J. P. 1996. Exceeding the dataflow limit via value prediction. In *29th International Symposium on Microarchitecture*.

PAI, V. S., RANGANATHAN, P., AND ADVE, S. V. 1997. The impact of instruction-level parallelism on multiprocessor performance and simulation methodology. In *3rd International Symposium on High Performance Computer Architecture*.

PATEL, S., EVERS, M., AND PATT, Y. 1998. Improving trace cache effectiveness with branch promotion and trace packing. In *25th International Symposium on Computer Architecture*.

RAJWAR, R. AND GOODMAN, J. R. 2001. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *34th International Symposium on Microarchitecture*.

RAKVIC, R., BLACK, B., LIMAYE, D., AND SHEN, J. P. 2002. Non-vital loads. In *8th International Symposium on High-Performance Computer Architecture*.

RANGANATHAN, P., GHARACHORLOO, K., ADVE, S. V., AND BARROSO, L. A. 1998. Performance of database workloads on shared-memory systems with out-of-order processors.

ROSENBLUM, M., BUGNION, E., HERROD, S. A., WITCHEL, E., AND GUPTA, A. 1995. The impact of architectural trends on operating system performance. In *15th Symposium on Operating Systems Principles*.

ROTH, A. AND SOHI, G. 2000. *Speculative Data-Driven Sequencing for Imperative Programs*. Tech. Rep. CS-TR-2000-1411, University of Wisconsin, Madison.

SASANKA, R., HUGHES, C. J., AND ADVE, S. V. 2002. Joint local and global hardware adaptations for energy. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems*.

SEMERARO, G., MAGKLIS, G., BALASUBRAMONIAN, R., ALBONESI, D., DWARKADAS, S., AND SCOTT, M. 2002. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *8th International Symposium on High-Performance Computer Architecture*.

SENG, J. S., TUNE, E. S., AND TULLSEN, D. M. 2001. Reducing power with dynamic critical path information. In *34th International Symposium on Microarchitecture*.

SODANI, A. AND SOHI, G. S. 1997. Dynamic instruction reuse. In *24th International Symposium on Computer Architecture*.

SPRANGLE, E. AND CARMEAN, D. 2002. Increasing processor performance by implementing deeper pipelines. In *29th International Symposium on Computer Architecture*.

SPRUNT, B. 2002. Pentium 4 performance-monitoring features. *IEEE Micro*, July.

SRINIVASAN, S. T., CHING JU, R. D., LEBECK, A. R., AND WILKERSON, C. 2001. Locality vs. criticality. In *28th International Symposium on Computer Architecture*.

SRINIVASAN, S. T. AND LEBECK, A. R. 1998. Load latency tolerance in dynamically scheduled processors. In *31st International Symposium on Microarchitecture*.

STEFFAN, J. G., COLOHAN, C. B., ZHAI, A., AND MOWRY, T. C. 2000. A scalable approach to thread-level speculation. In *27th International Symposium on Computer Architecture*.

TUNE, E., LIANG, D., TULLSEN, D. M., AND CALDER, B. 2001. Dynamic prediction of critical path instructions. In *7th International Symposium on High-Performance Computer Architecture*.

TUNE, E., TULLSEN, D., AND CALDER, B. 2002. Quantifying instruction criticality. In *11th International Conference on Parallel Architectures and Compilation Techniques*.

YI, J. J., LILJA, D. J., AND HAWKINS, D. M. 2003. A statistically rigorous approach for improving simulation methodology. In *9th International Symposium on High Performance Computer Architecture*.

ZAGHA, M., LARSON, B., TURNER, S., AND ITZKOWITZ, M. 1996. Performance analysis using the MIPS R10000 performance counters. In *Supercomputing '96*.

ZILLES, C. AND SOHI, G. 2001. Execution-based prediction using speculative slices. In *Proceedings of the 28th International Symposium on Computer Architecture*.

Received April 2004; revised June 2004; accepted June 2004